The ISC Int'l Journal of Information Security

January 2022, Volume 14, Number 1 (pp. 83–104)

http://www.isecure-journal.org

Business-Layer Session Puzzling Racer: Dynamic Security Testing Against Session Puzzling Race Conditions in Business Layer

Mitra Alidoosti [1,*], Alireza Nowroozi [2], and Ahmad Nickabadi [3]

[1] *Malek-Ashtar University of Tehran, Tehran, Iran.*
[2] *IRIB University of Tehran, Tehran, Iran.*
[3] *Amirkabir University of Tehran, Tehran, Iran.*

ARTICLE INFO.

Article history:
Received: February 10, 2021
Revised: June 30, 2021
Accepted: July 6, 2021
Published Online: September 6, 2021

Keywords:
Dynamic Testing, Vulnerability Analysis, Web Application, Business Process, Race Condition

Type: Research Article

doi: 10.22042/isecure.2021.272808.637

dor: 20.1001.1.20082045.2022.14.1.7.7

ABSTRACT

Parallel execution of multiple threads of a web application will result in server-side races if the web application is not synchronized correctly. Server-side race is susceptible to flaws in the relation between the server and the database. Detecting the race condition in the web applications depends on the business logic of the application. No logic-aware approach has been presented to deal with race conditions. Furthermore, most existing approaches either result in DoS or are not applicable with false positive. In this study, the session puzzling race conditions existing in a web application are classified and described. In addition, we present Business-Layer Session Puzzling Racer, a black-box approach for dynamic application security testing, to detect the business-layer vulnerability of the application against session puzzling race conditions. Experiments on well-known and widely used web applications showed that Business-Layer Session Puzzling Racer is able to detect the business layer vulnerabilities of these applications against race conditions. In addition, the amount of traffic generated to identify the vulnerabilities has been improved by about 94.38% by identifying the business layer of the application. Thus, Business-Layer Session Puzzling Racer does not result in DoS.

1 Introduction

Today, web applications play an essential role in our everyday life. They are used in all parts of our life, including education, entertainment, tax, purchase and bank transactions among others. Today, in order to provide complicated operations and proper user interface to the applications, complicated logic is added to client-side pages. Due to exchanging essential information, applications need to be secure and available (even at peak hours and during adversary attack) [1–12]. Race condition is a class of time-related vulnerabilities. Race condition occurs when access to a common variable by various processes is not managed correctly. The damage caused by race condition includes bypassing constraints of the application, like reusing voucher, privilege escalation and DoS. Detecting race condition in a web application is very difficult [13] and depends on the logic of the web application [14]. That is because the race condition is a probabilistic event which is state and time-related. For the race condition to occur, the application should be in a specific state and even the operations should be done in a minimal timeframe

* Corresponding author.
Email addresses: Alidoosti@mut.ac.ir, alirezanowroozi@iribu.ac.ir, Nickabadi@aut.ac.ir
ISSN: 2008-2045

known as "Race Window". The race condition yields unpredictable results and leads to an incorrect execution of the web application. Thus, detecting the race condition is necessary. Applications should be aware of the execution threads of the program and in cases where several threads use a common object in parallel the program should schedule the threads. In general, two types of race condition might occur in the application; server-side race and client-side race. The server-side race is one of the most common race conditions in the web applications occurring because of flaws in the web application's interaction with the database [15]. The client-side race occurs due to flaws in the server-client communication [16]. Research Gaps: No logic-aware approach has been proposed to detect synchronization problems in the web applications. The methods presented in [13, 14, 17–24] only detect parts of the sever-side race condition and are specific to a particular language. In addition, old test methods cannot detect the client-side race. Recently, some methods have been presented to detect parts of the client-side race conditions [25–36]. However, each of the above methods has its own shortcomings and false positive. In this study, a black-box approach called Business-Layer Session Puzzling Racer is presented for dynamic application security testing in the business layer. The proposed method detects the business-layer vulnerability of the application against session puzzling race condition. The designed security test scenarios are context-aware. Context-awareness indicates that the designed scenarios are aware of the business logic of the web application and intend to detect the business-layer vulnerabilities of the web application. The proposed method is independent of the technology used in the web application and detects vulnerabilities automatically. The innovations of this study are as follows:

- Classifying various session puzzling race conditions in the web applications;
- Defining various session puzzling race conditions existing in the web applications;
- Presenting the black-box approach for the dynamic application security testing of the web application to detect business-layer vulnerabilities against session puzzling race conditions.

2 Related Work

2.1 Detecting Server-Side Race Conditions in Web Applications

CompuRacer [13] has presented a systematic black-box method for detecting race condition in the web applications. The proposed method offers a list of common vulnerabilities of the race condition in applications and a test method for detecting them. CompuRacer forwards a set of HTTP requests to the application in parallel and detects vulnerabilities through analyzing the received HTTP responses. Palleri *et al.* [14] have proposed a dynamic method for detecting the race conditions originating from the interaction of the web application with the database. The proposed method detects vulnerabilities in two steps. In the first step, the report of the queries exchanged between the application and the database is stored. In the second step, the queries of the database are analyzed and the related queries are detected.

2.2 Detecting Client-Side Race Conditions in Web Applications

ARROW [35] has presented a static method to detect client-side race conditions. ARROW first extracts the "causal graph" of the application, which is tasked with modeling happens-before relationships between runtime events (for instance, DOM objects should be developed before invoking their event handlers). Then, ARROW specifies the "Def-use" relationships that would determine the variable/object codes and their use. "Def-use" relationships are detected through investigating HTML codes and Java Script codes and indicate the programmer's expectation of the execution sequence of the program. Wherever there is a difference between the "causal graph" and the "Def-use" relationships, the race condition is detected. RClassify [36] detects harmful race conditions on the client-side statically. In order to detect harmful race conditions, race condition alarms are first received as input. For each pair of events a and b, whose race condition alert is received, RClassify first executes event a and then event b. Then, it executes the events by reversing the order. If different results are achieved from the two executions, the variables of Java Script or HTML DOM or ... are different, which means that harmful race condition is detected. EventRaceCommander [37] has presented an automatic method to resolve client-side race conditions statically; in other words, it has presented a static approach for detecting AJAX race conditions. The proposed method resolves the race condition vulnerability in the Java Script web applications through controlling the events and postponing event handlers. One of the implemented policies is to postpone AJAX requests until the response of the first AJAX request is received. Adamsen [34] has presented a novel method for detecting the race conditions occurring during the initialization phase of JavaScript web applications. The proposed method detects vulnerabilities dynamically. Such events occur due to uncertainty in the order of executing the event handlers. The tools designed in this paper, known as INITRACER, can detect the events occurring in the initialization phase of the web

application. AJAXRACER [33] has presented a dynamic method to detect harmful AJAX race conditions. The proposed method has two steps. In the first step, the program is executed and event handlers with common AJAX responses are detected. A graph is provided for each event, which carries information about the event and its invocation. For each two events with common Ajax response, two execution modes are studied. First, the events are executed serially without any time interference. In the second mode, the events are executed with time interference. Finally, the results are compared. If the results are different, the application will be vulnerable to the race condition. Petro *et al.* [28] formally defined the relationship with a happens-before order for the first time and designed WebRacer based on these concepts to detect the client-side race dynamically. Mutlu *et al.* [32] have presented a method for detecting harmful race conditions on the client-side (AJAX race) dynamically. Harmful race conditions result in a persistent browser state or server change. It detects race conditions that change sensitive variables (like client-side cookies, local storage, session storage and HTML DOM). This detection method checks the traffic obtained from the execution of the application and analyzes whether or not the sensitive variables receive different values at different executions. If different values are received a race condition is detected. EventRacer [31] has proposed a dynamic method for detecting race conditions in JavaScript applications with the event concept. EventRacer detects vulnerabilities, considering the events existing in the application and defining happens-before relationships and a vector clock for the events.

3 Defining Race Condition and Types of Race Condition

In the web applications, the race condition is divided into two categories; server-side race and client-side race. Server-side race conditions occur when different processes access common data without proper synchronization and at least one of them intends to write on the common data. The race condition originates from the bugs of programmers due to improper synchronization [17, 18]. Server-side race condition occurs when the operations are not atomic [21]. Client-side race condition occurs because many of the JavaScript web applications employ AJAX for server-client communication. In addition, asynchronous requests (asynchronous scripts, requests for receiving external resources etc.) result in client-side race condition.

3.1 Defining Race Window

Race window is the duration of applying the race attack vector to the application. If the attack vector occurs in the time window and the application is vulnerable, the attack would be successful. "Server-side race window" is defined as follows.

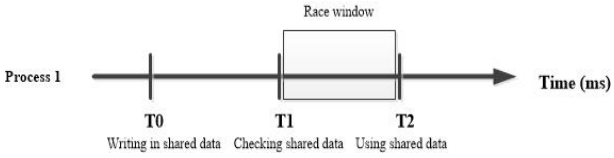

Figure 1. Race window

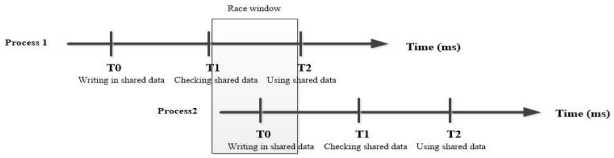

Figure 2. Condition required for server-side race to occur in web application

Definition 1 (Server-Side Race Window). Race window in a process is the interval between checking shared data and using them (performing operations).

$$T_1, T_2 \in Time, T_1 < T_2$$

RW = { [T1, T2] in Process1 | the time between checking shared data and using them (acting by them) in a process }

Figure 1 shows the race window. If a process changes the shared data during the race window, race condition will occur in the web application. Figure 2 shows the condition required for the race condition to occur.

3.2 Detected Race Conditions

One of our innovations is stating the varieties of session puzzling races in the web application. We show varietes of session puzzling race conditions in the web application in Table 1. Naming and categorizing are all our innovations. We describe them in details as follows.

3.3 Session Puzzling Race

"Session puzzle" refers to a type of web application vulnerability that can be detected and misused through applying the race vector at the application level. Such attack vectors have no malicious input. In other words, the session puzzle occurs due to the uncontrolled creation and population of sessions or using identical session-identifier value at the entry points of the application [26]. The only condition for the existence of vulnerability of "session puzzle" is that there are at least two entry points of the application with identical session-identifier value and the attack vector is

applied in the race window. The objectives of the adversary in the session puzzling race are as follows:

- Bypassing authentication mechanism and impersonating legal users;
- Privilege escalation of the malicious user;
- Passing the conditional steps in multi-step processes even if proper constraints are considered in the code level.

Various types of "Session Puzzling Races" are described as follows.

3.3.1 Log-in Session Puzzling Race

The guest user tries to log-in to one or more user accounts in parallel. If the sessions are not synchronized properly in the server, parallel log-ins to multiple user accounts simultaneously affect each other. For two different user accounts, that raises puzzled sessions, followed by confidentiality violation. Simultaneous log-in to one user account in parallel violates integrity because parallel operations are performed via two different sessions on one user account. These operations might interfere with each other. Simultaneous log-out is also studied, but no example is found in the web applications. Log-in Session Puzzling Race is shown in Figure 3. It should be noted that Alice performs the first process and Bob performs the second process. According to Figure 3, the first process (P1) and the second process (P2) can be described as follows:

P1: WRITE (P1, usernameAlice) → CHECK (P1, usernameAlice) → ACT (P1, log-in) or USE (P1, log-in)

P2: WRITE (P2, usernameBob) → CHECK (P2, usernameBob) → ACT (P2, log-in) or USE (P2, log-in)

The race window of this attack vector can be described as follows:

RW= { [T1, T2] in P1 | T1: when P1 completes WRITE (P1, usernameAlice) , T2: before P1 starts ACT (P1, log-in) or USE (P1, log-in) }

The occurrence condition of "Log-in Session Puzzling Race" is as follows:

WRITE (P2, usernameBob) occurs in Race Window

3.3.2 Authentication-Bypass Session Puzzling Race

This attack vector can bypass the authentication mechanism by creating a race condition and using the "session puzzling" attack vector. In order to misuse this type of vulnerability, all entry points of the web application with the same session-identifier value are detected. In other words, if the application creates the session-identifier value based on username, the page with user authentication and the page without user authentication that only receives the username

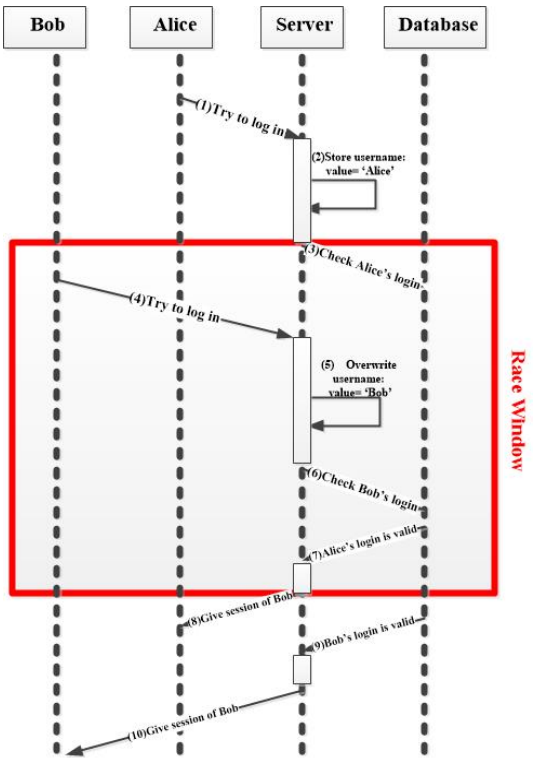

Figure 3. Log-in Session Puzzling Race

can have the same session-identifier value. In this attack, user1 is authenticated at one entry point and at another entry point that can be accessed by the public, and only a username is received, user2 enters his username. Here, since the session identifier of user2, either authenticated or not authenticated, is the same, user1 can see the profile of user2. Therefore, we have logged in to the profile of user2 without authentication.

"Authentication-Bypass Session Puzzling Race" is shown in Figure 4. According to Figure 4, the first and second processes may be described as follows:

P1: WRITE (P1,usernameAlice)

P2: WRITE (P2,usernameBob)

The race window of this attack vector can be described as follows:

RW= { [T1, T2] in P1 | T1: when P1 completes writing (after log-in) , T2: before P1 logs out }

The condition for "Authentication-Bypass Session Puzzling Race" to occur is as follows:

WRITE (P2,usernameBob) occurs in Race Window

3.3.3 User-Impersonation Session Puzzling Race

This attack vector is like "authentication-bypass session puzzling race" with the difference that both en-

Table 1. Varieties of Session Puzzling Race Conditions in web application

Server-Side Race	Session Puzzling Race	Log-in Session Puzzling Race
		Authentication-Bypass Session Puzzling Race
		User-Impersonation Session Puzzling Race
		Privileged-Escalation Session Puzzling Race
		Flow-Enforcement-Bypass Session Puzzling Race

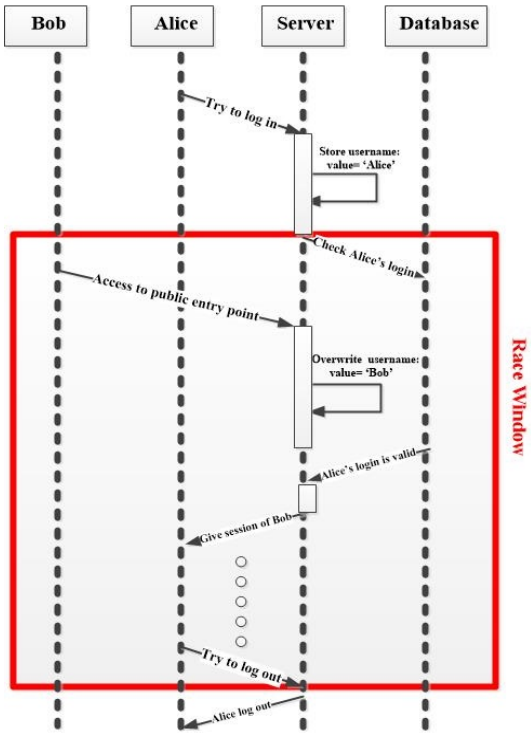

Figure 4. Authentication-Bypass Session Puzzling Race

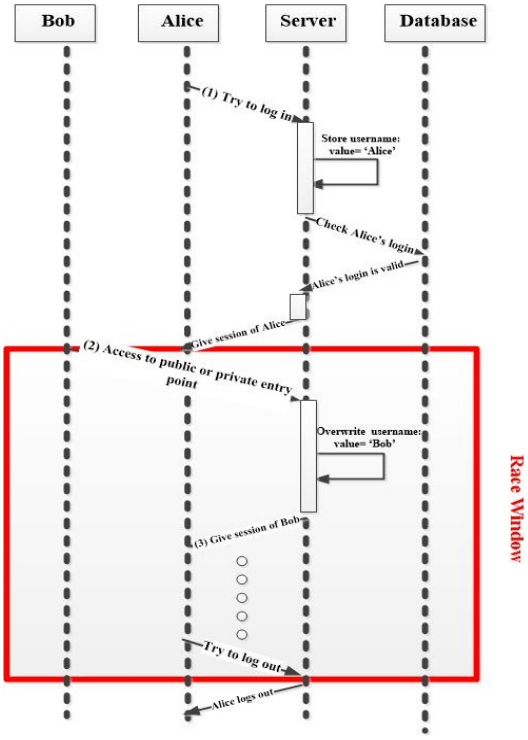

Figure 5. User-Impersonation Session Puzzling Race

try points with the same session-identifier value have an authentication mechanism. The second difference is that the value overwritten in the session identifier should belong to a valid user. "User-Impersonation Session Puzzling Race" is shown in Figure 5. According to Figure 5, the first and second processes can be described as follows:

P1: WRITE (P1, usernameAlice)

P2: WRITE (P2, usernameBob)

The race window of this attack vector can be described as follows:

RW= { [T1, T2] in P1 | T1: when P1 completes writing (after log-in) , T2: before P1 logs out }

The condition for "User-Impersonation Session Puzzling Race" to occur is as follows:

WRITE (P2,usernameBob) occurs in Race Window

3.3.4 Privilege-Escalation Session Puzzling Race

Since user roles and privileges are usually stored in the session-identifier value, "session puzzling" escalates the privilege by changing the session-identifier value, and therefore the user can access the contents to which he had not access before.

Since the client does not usually initialize the privileges, finding the entry point, which provides the possibility to change the value of the role-related session-identifier, is difficult. One of these entry points is role-specific contents which might be able to change their session-identifier value.

In order to use session puzzling for privilege escalation, the adversary should be authenticated (or use authentication-bypass mechanisms) before access to the entry points of the application whose role-related variable has the same session storage with the entry point with a lower privilege. This may stem from

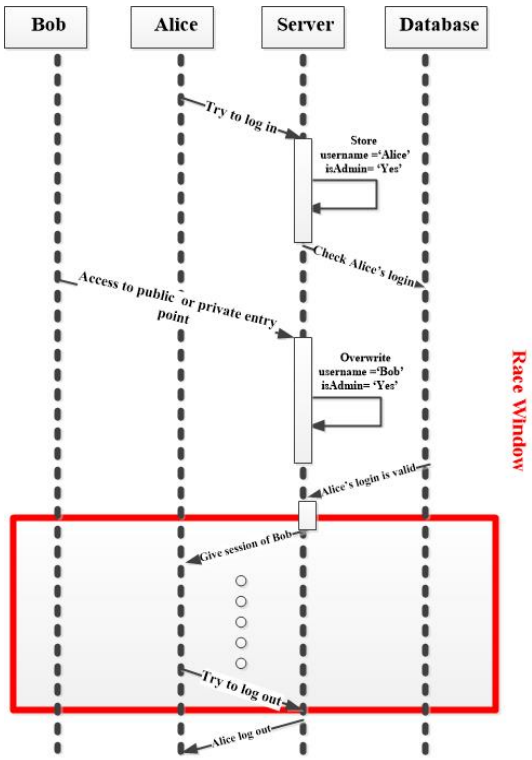

Figure 6. Privilege-Escalation Session Puzzling Race

the implementation flaws or the existence of limited points in the application to which the privilege is not applied correctly.

"Privilege-Escalation Session Puzzling Race" is shown in Figure 6. According to Figure 6, the first and second processes can be described as follows:

P1: WRITE (P1, usernameAlice and isAdminNo)

P2: WRITE (P2, usernameBob and isAdminYes)

The race window of this attack vector can be described as follows:

RW= { [T1, T2] in P1 | T1: when P1 completes writing (after log-in) , T2: before P1 logs out }

The condition for "Privilege-Escalation Session Puzzling Race" to occur is as follows:

WRITE (P2, usernameBob and isAdminYes) occurs in Race Window

The other way to escalate privileges is as follows: The administrator tries to change roles and privileges of some users and administrators in parallel. If the sessions are not synchronized properly in the server, parallel requests simultaneously affect each other and cause escalation of the privilege of the user. Therefore, he can access the data which he could not before. The administrator performs this vulnerability and the tester is not eager to check this because the administrator user can apply more harmful operations. Indeed, it can be considered as an internal attack vector for applying harmful operations without formal privileges and without an operation report of the user being recorded.

3.3.5 Flow-Enforcement-Bypass Session Puzzling Race

One of the most interesting capabilities of the session puzzling is the flow enforcement in multi-step processes. Some conditional multi-step processes are as follows: password recovery process (the conditional step is responding the question in which the adversary should know the answer that the user has given to the question in the registering process), financial transactions process (entering specifications of the user's bank account), and permission grant processes (authenticating the user for completing the operation again). In order to enforce the flow by session puzzling, at least two multi-step processes with identical flow flags should be executed simultaneously. It should be mentioned that the counter for the steps of these two processes should be the same. For instance, in order to bypass the step of entering the information of the bank account in the financial transaction process, the user registration, which is a multi-step process (it is a process without conditional step), is executed simultaneously. If the flags of these two processes are the same, the conditional step (entering information of the bank account) can be bypassed.

In general, both multi-step processes (one process must be conditional and the other unconditional) should be executed simultaneously, but the adversary should execute the unconditional multi-step process first and then use its flags to bypass the conditional step in the conditional multi-step process. In addition, the counter for the steps of the two processes should be the same. A common multi-step process in unconditional and unrestricted applications is user registration.

"Flow-Enforcement-Bypass Session Puzzling Race" is shown in Figure 7. The steps of "Flow-Enforcement-Bypass Session Puzzling Race" are as follows:

Assumption 1: The user registration process and password recovery process have the same flags, and in addition, the lengths of both processes are the same.

Assumption 2: Alice and Bob have the same level of access.

(1) The first process consists of the user's registration process in order to create and initialize the flags and stop them until the last step (P1:WRITE (P1, flags));
(2) The second process is to start the password recovery process (P2: WRITE (P2, flags));
(3) Continue with the user registration process and

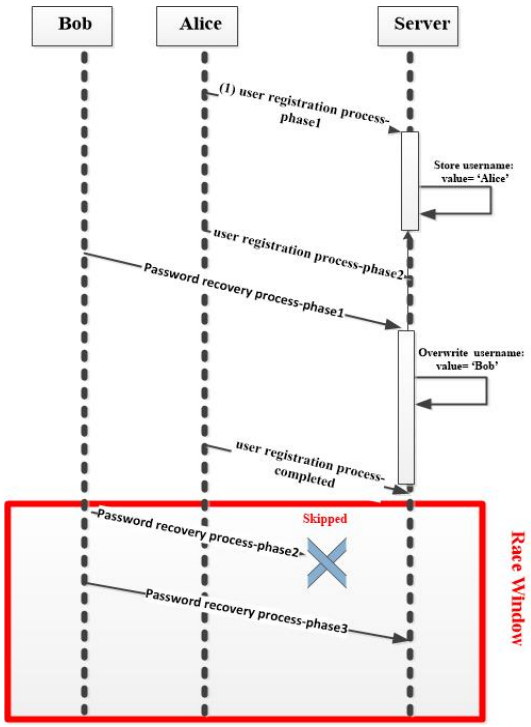

Figure 7. Flow-Enforcement-Bypass Session Puzzling Race

complete it (P1:WRITE (P1, flags));

(4) The user registration process flags are used in this process to pass the conditional step in the recovery password process. Therefore bypassing the conditional step in the recovery password process (by changing the URL to success status) and finally Bob can change the password in the password recovery process.

According to Figure 7, the first and the second processes are unconditional multi-step process and conditional multi-step process respectively. They can be described as follows:

P1: WRITE (P1,flags)

P2: WRITE (P2,flags)

P1: WRITE (P1,flags)

The race window of this attack vector can be described as follows:

RW= { [T1, T2] in P2 | T1: when P2 starting, T2: before P2 reaches to sensitive phase }

The condition for "Flow-Enforcement-Bypass Session Puzzling Race" to occur is as follows:

WRITE (P1, flags) occurs in Race Window

4 Business-Layer Session Puzzling Racer

In order to evaluate security of the application against race conditions, Business-Layer Session Puzzling Racer is proposed. Business-Layer Session Puzzling Racer employs dynamic web application security testing in the business layer to detect the vulnerability of the application against race condition attacks. Figure 8 depicts the Business-Layer Session Puzzling Racer overview. Business-Layer Session Puzzling Racer is situated as a proxy between the web application and the web server. Business-Layer Session Puzzling Racer comprises three main steps as follows:

(1) Finding the business processes of the application;
(2) Detecting processes prone to race condition in the business layer of the application;
(3) Applying race condition test scenarios in the business-layer of the web application and evaluating the results.

The normal user scans the web application sequentially and traffic of the normal user is stored. Business-Layer Session Puzzling Racer first extracts the user navigation graph. Then, from the generated graph, the business processes of the web application are extracted. Then, the business processes prone to race condition attack are detected. Finally, the race condition testing scenario is applied to the business layer according to the detected critical processes. Figure 9 shows the Business-Layer Session Puzzling Racer architecture. Each step is described in details hereunder.

4.1 Finding Business Processes of the Web Application

In order to detect the business process of the web application, the user navigation graph should be extracted first. Then, the generated graph should be used to detect the business processes. We discussed extracting business processes in the web applications in our previous work BLProM [1, 2] in details. Figure 10 depicts the steps involved in BLProM.

4.1.1 Extracting User Navigation Graph

Business-Layer Session Puzzling Racer first extracts the user navigation graph from the stored traffic. Figure 3 depicts the steps involved in extracting the user navigation graph. The steps are as follows:

(1) Preprocessing the raw input data;
(2) Detecting the web pages of the web application existing in the stored traffic;
(3) Clustering the web pages;
(4) Extracting the user navigation graph.

Preprocessing raw input data. Business-Layer Session Puzzling Racer sanitizes data to eliminate unnecessary items. In this study, only HTTP requests and HTTP responses are needed; nonetheless, the

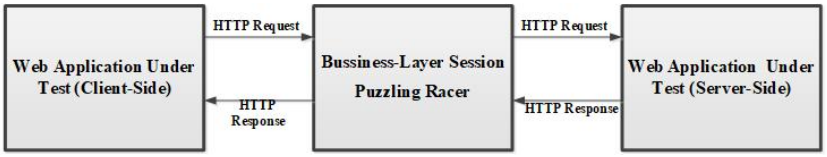

Figure 8. Business-Layer Session Puzzling Racer overview

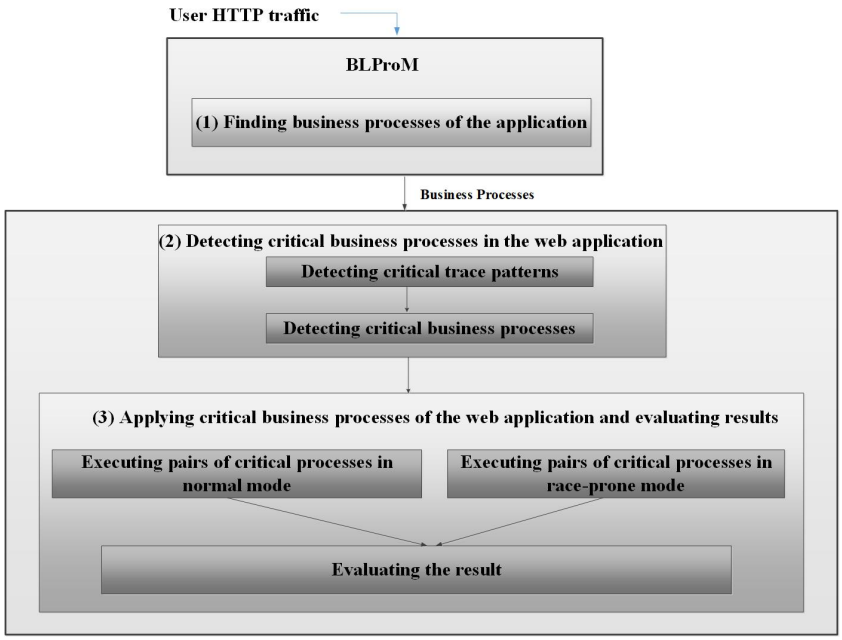

Figure 9. Business-Layer Session Puzzling Racer overview

HTTP responses having a successful status code 200 are needed. Business-Layer Session Puzzling Racer eliminates the responses having unsuccessful codes and their corresponding requests. In addition, in this study, only the GET and POST requests are needed.

Detecting web-application pages existing in the stored traffic. Each web page of the web application can be modeled as a pair (request set, response set). The request set is a set of HTTP requests sent for loading the page. The response set is a set of HTTP responses sent for loading the page.

A user's stored traffic contains the main HTTP requests for loading the webpage as well as the secondary requests meant to load files of the page. To model the pages existing in the HTTP traffic as a pair (request set, response set), first the main requests in the HTTP traffic are detected. The HTTP traffic based on the main detected requests is then divided into blocks. Each block includes the main request and the HTTP traffic between the two main requests. All the requests and responses in the same block represent the request and response sets of a page. Business-Layer Session Puzzling Racer considers a request as the main request if its referer header is different from the referer of the next request. In addition, the first request existing in the traffic is considered as the main request because the first traffic does not have a referer. It should be mentioned that the referer header is a header of the HTTP request that represents the URI address of the previous page visited by the user.

Thus far, web-application pages have been modeled as a pair (request set, response set). The main response in the response set of each page should also be detected. The main responses have a content-type header with a text/html value. The main response is of text/html type. Furthermore, if the last response of the traffic is the main response, the last HTTP request will be considered as the main request. Therefore, each page of the web application is modeled as a pair (request set, response set), and the main requests and responses are marked.

Clustering web pages. In this step, Business-Layer Session Puzzling Racer clusters the web pages. The purpose of clustering is to place similar web pages in one cluster. Each cluster represents one node of the user navigation graph. Therefore, the infinite development of the graph is avoided.

Each pair (request set, response set) shows one page of the web application. To extract the optimal user

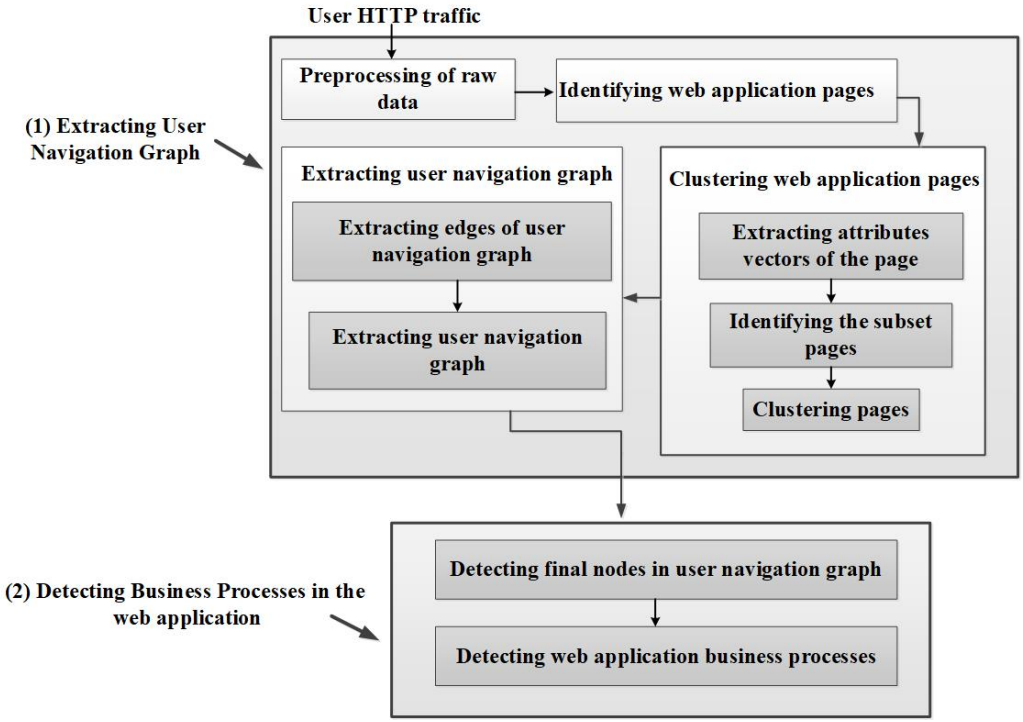

Figure 10. BLProM: Black-box approach to detect web application business process

navigation graph, similar pages should be identified and subsequently clustered. In the user navigation graph, the nodes represent the unique pages of the web application and the edges serve as connection between the pages. In our context, two pages are considered to be similar when the user can perform similar actions on them. For instance, consider two pages with a button. The button on the first page is "continue," and the button on the second page is "save." These two pages are different because the user performs different actions on them.

Because the final goal of Business-Layer Session Puzzling Racer is to detect vulnerabilities, clustering identifies similar pages based on the goal of Business-Layer Session Puzzling Racer. The input fields and hyperlinks facilitating interaction with the application are critical points in detecting vulnerabilities. Therefore, HTML elements that have critical points are considered for detecting similar pages. These elements include buttons, inputs, and anchors existing in each page. In addition, in the web-application pages, the position of images is related to the logic of the page. Therefore, the position of image is another important element in detecting similar pages.

Definition 2 (Similar Pages). Two pages are considered to be similar if the user is able to perform the same actions on them and the positions of important HTML elements in the pages are the same. Important HTML elements include buttons, inputs, anchors, and images existing in each page.

Definition 3 (Similar Pages). Pages whose structures are the subsets of another page in terms of important HTML elements are similar to the reference page.

BLDATS identifies similar pages by comparing the feature vectors of two pages with each other. The feature vector of each page is the DOM path of important HTML elements existing in the page. Business-Layer Session Puzzling Racer considers every two pages with the same feature vector as similar pages and puts them in one cluster. The algorithm used to identify similar pages is described in Section 5.1.

Extracting user navigation graph. In this step, Business-Layer Session Puzzling Racer first extracts the graph edges that connect clusters. Each cluster represents a unique web-application page. Each cluster has a set of similar pages, each of which containing a URI and a referer field. Therefore, each cluster has a set of URIs and a set of referers, both of which are associated with the pages existing in that particular cluster. To find graph edges, the URI set of each cluster is compared to the referer set of other clusters. If they have at least one member in common, they will connect to each other. Assume that the URI set of cluster C1 shares members with the referer set of clusters C2 and C3. In that case, one can move from C1 to C2 and C3 (C1 $\rightarrow$ C2 and C1 $\rightarrow$ C3); in other words, C1C2 and C1C3 are graph edges.

Definition 4 (User Navigation Graph). This graph is represented by tuple $\langle C_0, C, E\rangle$. Here, C de-

notes the set of graph nodes, C_0 the initial (first) node of the graph, E the set of graph edges.

$$C_0 \subseteq C; E \subseteq C \times C$$

4.1.2 Detecting Business Processes in Web Application

In this step, Business-Layer Session Puzzling Racer first identifies the final nodes in the user navigation graph, and it then detects business processes. The steps to detect business processes are depicted in Fig. 4. We now define the web-application process, final node, and business process.

Definition 5 (Web-Application Process). A process P in a web application is a sequence of edges in the user navigation graph, such as $E_1, E_2, \ldots, E_k$ that

$$E_i \in E \ ; \ E_i = C_{i-1}C_i.$$

Definition 6 (Final Node in the User Navigation Graph). The final node F is the node such that when a web application reaches it, then the business process is completed.

By investigating HTTP responses, the final node can be identified. For instance, when a product is purchased, an expression such as "thank you for your purchase" is shown. By identifying a set of such expressions and searching for these expressions in the response messages, the final node can be determined. The keywords used to identify the final node are thanks, congratulations, successfully, log off, and search results among others. In addition, some buttons are good indicators for identifying the final node. The page after save button, the page after create button, and the page after submit button are some examples in this context.

Definition 7 (Business Process in the Web Application). A business process BP in the web application is a process that meets at least one of the following requirements:

(1) The beginning node of the process is the initial node of the user navigation graph (C0), and the ending node of the process is the final node of the user navigation graph (F);
(2) If the process passes the beginning node once again and if the process length is greater than 2. (If in a process, the user returns to the beginning node and if the length of the loop is greater than 2, it is a business process).

4.2 Detecting Critical Business Processes in the Web Application

The proposed method uses the outputs of the BL-PRoM to perform the black-box test on the business layer of the web application and takes the identified business processes as input. Business-Layer Session Puzzling Racer goes through the following two stages:

(1) Identifying race condition attack-prone processes in the business layer of the application;
(2) Performing the black-box test on the business layer and checking the results.

In order to identify the processes susceptible to the race condition attack, or what we herein refer to as critical processes, one must identify the business processes whose pages exhibit particular trace patterns. The trace patterns corresponding to different race conditions are discussed here. These processes are vulnerable to the race condition attack. Algorithm 1 shows the pseudocode for identifying the critical processes (vulnerabilities to race condition attack in the application).

Algorithm 1 CriticalProcessDetection

Input: : TrP as a set of trace pattern
BP as a set of business processes in the web application
Output: CrP as a set of critical process

```
1: procedure CriticalProcessDetection
2:     Algorithm CriticalProcessDetection
3:     Begin
4:     Let CrP= Ø; //set of web application critical processes
5:     Let i=0; //counter for business processes in BP
6:     Let j=0; //counter for web pages in a business processes
7:     for each business process in BP do
8:     for each web pages in the business processi do
9:       if (web pagej has a trace pattern in TrP)
10:      CrP ← business processi
11:      done;
12:      j=j+1;
13:      done;
14:      i=i+1;
15:      done;
16:      end
17: end procedure
```

4.2.1 Required Definitions

HTML-element-start[e,o,i]: Marks the start of an HTML element, where e is the event index, o is the DOM element along with its attributes, and i contains some information on the HTML element (e.g. VISIBLE refers to visibility of the element while WRITABLE declares that the element is neither readable nor inactive).

$\{...\}_{ji}$: shows the combination of the HTML element for the indicated numbers, where i and j are the minimum and maximum counter for repetitions, respectively.

4.2.2 Detecting Critical Processes of "Log-in Session Puzzling Race"

Under this race condition, the critical process consists of logging in the user account. The following trace pattern is defined to identify the business processes that log-in the user account.

Figure 11 shows the trace pattern characterizing a user log-in page. According to this pattern, any page of the application that contains an HTML form with two writable input elements and one button element is recognized as a user log-in page. In the Log-in Session Puzzling Race condition, the shared data refers to the user name. The writing in the user name occurs in a page of the application with the trace pattern shown in Figure 11, where all input fields are filled in. The use name checking operation and then acting operation occurs once the log-in button in the trace pattern described in Figure 11 is clicked. By definition, the race condition time window spans between the writing and acting of the user name.

4.2.3 Detecting Critical Processes of "Authentication-Bypass Session Puzzling Race"

Critical processes in "Authentication-Bypass Session Puzzling Race are all business processes with the authentication mechanism (log-in processes shown in Figure 11) and business processes without authentication that can be accessed by public. It should be mentioned that the processes which can be accessed by public should receive a username at the entry point; we call these process public-entry-point processes. For example, password recovery and user registration processes are public-entry-point process. According to Figure 12, pages existing in these processes have an HTML button or clickable input labeled as "recovery password", "register", "forgot password" and "sign up". If such buttons exist in the page of a process, the process can be public-entry-point process.

4.2.4 Detecting critical process of "User-Impersonation Session Puzzling Race"

The trace pattern for detecting the critical process of "User-Impersonation Session Puzzling Race" is similar to "Authentication-Bypass Session Puzzling Race".

4.2.5 Detecting critical process of "Privilege-Escalation Session Puzzling Race"

In order to detect the critical process of "Privilege-Escalation Session Puzzling Race", "write parameter" and "write process" should be defined first.

Definition 8 (Write Parameter). Assume that the application W includes a set of web pages WP where each web page is a pair of (REQ request set, RESP response set). Each request using GET or POST method can have parameters that are called Parameters and are represented with PP. Parameter PP is called the write parameter if its value is stored in a table in the dataset for each request sent to load the page.

Definition 9 (Write Process). The write process is the business process where at least one of the web pages of the process has the "write parameter".

The pseudo-code for extracting the "write processes" in the web application is shown in Algorithm 2. For all existing business processes in the web application (line 5), all pages of the process are checked (line 6); if at least, one of the pages has the write parameter (line 7), the business process of interest is the write process (line 8).

Algorithm 2 ExtractWriteProcess

Input: : BP as a set of business processes of the web application
Output: CrP as a set of web application Write processes

```
1: procedure ExtractWriteProcess
2:     Algorithm ExtractWriteProcess
3:     Begin
4:     Let CrP= Ø; //set of web application critical processes
5:     Let i=0; //counter for business processes in BP
6:     Let j=0; //counter for web pages in a business processes
7:     for each business process in BP do
8:     for each web pages in the business processi do
9:       if (web pagej has Write Parameter)
10:      CrP ← business processi
11:      GOTO here;
12:      done;
13:      j=j+1;
14:      done;
15:      here:
16:      i=i+1;
17:      done;
18:      return CrP;
19:      end
20: end procedure
```

4.2.6 Detecting Critical Process of "Flow-Enforcement-Bypass Session Puzzling Race"

The critical processes of "Flow-Enforcement-Bypass Session Puzzling Race are multiple step processes which are limited (sensitive multi-step processes) and the processes which are not limited (simple multi-step processes).

Definition 10 (Sensitive Multi-Step Process). "Sensitive multi-Step process" is the "write process" which performs sensitive operations on the system or the user account and includes qualifying steps to

{⟨ Log-inPattern ⟩ := {⟨ HTML-element- start[ej,oinput,VISIBLE,WRITABLE] ⟩} 2n ⟨ HTML-element- start[ek,obutton | Oa,VISIBLE,CLICK, "log" | "sign"]⟩ n=2 }

Figure 11. Trace pattern of Log-in

{⟨ public-entry point Pattern ⟩ := ⟨ HTML-element- start[ek,obutton | Oa,VISIBLE,CLICK, "recovery password"| "forgot password"| "register"| "sign up"| "New Customer"]⟩}

Figure 12. Trace pattern of public-entry point

check if the initiating entity has the proper identity or permission.

It should be mentioned that the "sensitive multi-step process" has a minimum length of 2.

Definition 11 (Simple Multi-Step Process). "Simple multi-Step processes" is the "write process" whose length is at least 2 and which is not a "sensitive multi-step process".

The pseudo-code for detecting the critical processes of "Flow-Enforcement-Bypass Session Puzzling Race" is shown in Algorithm 3 Lines 9 to 13 detect "sensitive multi-step processes" in the web application. The common sensitive multi-step processes of the application include password recovery, payment process and the process which required authentication for the rest of the operations. Therefore, when write operations have a minimum length of 2 (line 9) and one of the pages of the process has at least one of "recovery password", "forgot password", "cash" and "payment" keywords, or at least two pages of the process have Log-in-DOM (line 10), the mentioned process is a sensitive multi-step process (line 11).
Lines 14 to 18 define the race window of the "sensitive multi-step processes". The race window of the mentioned processes spans from the start of the process (finishing step 1) to the qualifying step (the step where there is at least one of the keywords) (line 16).
Lines 20 to 25 detect the "simple multi-step processes". Each write process which has a minimum length of 2 (line 21) and it is not a "sensitive multi-step process" (line 22) is a "simple multi-step process" (line 23).

4.3 Applying Critical Business Processes of the Web Application and Evaluating Results

At this stage, Business-Layer Session Puzzling Racer applies to the identified critical processes toward the web application and the results are checked to detect the possible vulnerabilities of the application. In general, the identified critical processes were invoked by two threads in both "normal" and "race condition-prone" modes. In the "normal" mode, the two threads were invoked successively, where the second thread was only invoked once the first thread came to completion. In the "race condition-prone" mode; however, the second thread invoked at some time within the time window of the first thread. Then we compare the outputs of critical process in the two modes to check for potential vulnerabilities of the application. Later, this stage of Business-Layer Session Puzzling Racer is thoroughly explained for each of the mentioned modes. A pseudo code for identifying race conditions are given in Algorithm 4. Of course, there are differences in the details for each type of the race mode.

4.3.1 Detecting "Log-in Session Puzzling Race"

The pseudo-code for detecting the vulnerability of web application against "Log-in Session Puzzling Race" is shown in Algorithm 5. For each log-in process (line 3), two threads called the thread1 with CR user account and thread2 with user account CR' (line 4 and 5) are created. The first and the second threads were first executed in race-prone mode and then in normal mode. In race-prone mode thread2 is executed in the race window of the thread1 (line 7) and then the final node of second thread is captured and saved to S1 (line 8). Then two threads are executed in normal mode; first thread2 is executed completely then thread1 is executed (line 11) and the final node of thread2 is captured and saved in S2 (line 12). If S1 and S2 are not same (line 13) the application is vulnerable because both processes with different credentials log-in to the same account.

4.3.2 Detecting "Authentication-Bypass Session Puzzling Race"

After detecting the critical processes, the log-in process is executed with the identity of user1 and then the public-entry-point process is continued until it reaches the step to enter the username and username of user2. In other words, the second process (the public-entry-point process) is executed in the race window of the first process (log-in process). If the first process is now logged in to user 2 profile, it means that we manage to bypass the user 2 authentication mechanism. It is explained by the fact that the web application uses identical session-identifier values for both authenticated and non-authenticated users.
The pseudo-code for detecting the vulnerability of web application against "Authentication-Bypass Session Puzzling Race" is shown in Algorithm 6 which

Algorithm 3 Flow-Enforcement-Bypass-Session-PuzzlingRaceDetection

Input: : WrP as a set of write processes of the web application
LGP as a set of web application Log-in processes KEYWORD1={"recovery password", "forgot password", "cash", "payment", ...}
KEYWORD2={"register", "sign up", "New Customer", ...} **Output:** Detecting if the web application is vulnerable or not

```
1: procedure FLOW-ENFORCEMENT-BYPASS-SESSION-PUZZLINGRACEDETECTION
2:     Algorithm Flow-Enforcement-Bypass-Session-PuzzlingRaceDetection
3:     Begin
4:     Let SenP= Ø; //set of web application sensitive multi-step processes
5:     LetSimP= Ø; //set of web application simple multi-step processes
6:      let i=1; // counter for write processes in WrP
7:      let j=1; // counter for web page in BP
8:      let k=1; // counter for simple multi-step processes in SimP
9:      let l=1; // counter for sensitive multi-step processes in SenP
10:     // detecting sensitive multi-step processes in the web application
11:     for each write process in WrP which its length ¿=2
12:     if (at least one of web pages in the write processi has KEYWORDS1 or at least two pages in the write processi has log-in DOM)
13:     SenP ← write processi;
14:     i++;
15:     done;
16:     //detecting race window of sensitive multi-step processes
17:     for each sensitive multi-step processes in SenP
18:     race window of sensitive multi-step processl is from second webpage of the process (finishing step 1) to the qualifying step (the step which has at least one of the KEYWORDS1)
19:     l++;
20:     done;
21:     let i,j=0;
22:     // detecting simple multi-step processes in the web application
23:     for each write process in WrP which its length ¿=2
24:     if (write processi is not a sensitive multi-step process)
25:     SimP ← write processi;
26:     i++;
27:     done;
28:     end
29: end procedure
```

Algorithm 4 RaceConditionDetection

Input: : CrP as a set of web application critical processes
Race windows (RW) of all critical processes in CrP **Output:** Detecting if the web application is vulnerable or not

```
1: procedure RACECONDITIONDETECTION
2:     Algorithm RaceConditionDetection
3:     Begin
4:      let i=1; // counter for critical processes in CrP
5:      for each critical process in CrP do
6:       let thread1 , thread2= critical processi;
7:      // execute thread1 and thread2 in "race-prone mode" and save results
8:       //ignore changes in race-prone mode and return threads in their initial node
9:      // execute thread1 and thread2 in "normal mode" and save results
10:      // analyzing results for detecting web application vulnerabilities
11:     i=i+1;
12:     done;
13:     end
14: end procedure
```

is derived from the pseudo-code of Algorithm 4. for each log-in process is called the first thread (lines 5), and for each public-entry- point process which is called the second thread (line 7), the first and the second thread are executed such that the write operation (entering username and sending it) of the second thread occurs in the race window of the first thread (line 9). Then the current page of the first thread is refreshed (line 10). If the current page of the first thread is different before and after being refreshed (line 11), the application is vulnerable.

4.3.3 Detecting "User-Impersonation Session Puzzling Race"

The approach for detecting "User-Impersonation Session Puzzling Race" is similar to "Authentication-Bypass Session Puzzling Race" that is shown in Algorithm 6.

4.3.4 Detecting "Privilege-Escalation Session Puzzling Race"

The pseudo-code for detecting the vulnerability of the web application against "Privilege-Escalation Session Puzzling Race" is shown in Algorithm 7 that is derived from pseudo-code of Algorithm 4. For each "log-in process" which is called as the first thread (line 4 and 5) and for each "write process" that is called the second thread (line 6 and 7), the first and the second threads are executed such that the write operation of

Algorithm 5 Log-in-Session-Puzzling-RaceDetection

Input: : CrP as a set of web application critical processes
At least two valid credentials (CR, CR')
Race windows (RW) of all critical processes in CrP
Output: Detecting if the web application is vulnerable or not

```
1: procedure Log-in-Session-PuzzlingRaceDetection
2:     Algorithm Log-in-Session-PuzzlingRaceDetection
3:     Begin
4:      let i=1; // counter for critical processes in CrP
5:      for each critical process in CrP do
6:      let thread1 = critical processi with credential CR;
7:       let thread2 = critical processi with credential CR';
8:      // execute thread1 and thread2 in "race-prone mode" and save results
9:      execute thread1 and thread2 so that WRITE(thread2, Content) in RWthread1;
10:     S1=screenshot of final node of thread2
11:      //ignore changes in race-prone mode and return threads in their initial node
12:     // execute thread1 and thread2 in "normal mode" and save results
13:     execute thread2 completely then execute thread1 completely
14:     S2=screenshot of final node of thread2
15:      if( S1 not equal S2)
16:      critical processi is vulnerable to Log-in-Session-Puzzling Race;
17:     i++;
18:     done;
19:     end
20: end procedure
```

the second thread occurs in the race window of the first thread (line 8), and then the current page of the first thread is refreshed (line 9). If the current page of the first thread before and after being refreshed is different (line 10), the application is vulnerable.

4.3.5 Detecting "Flow-Enforcement-Bypass Session Puzzling Race"

Algorithm 8 shows vulnerable processes to "Flow-Enforcement-Bypass Session Puzzling Race". For each "simple multi-step process", known the first thread (line 4, 5), and for each "sensitive multi-step process", known as the second thread (line 6, 7), if thread1 and thread 2 have the same length (line 8), the first thread is executed before reaching the last step (line 9) and the second thread is executed until reaching the sensitive step (line 10). The first thread is completed in the race window of the second thread (line 11). If the second thread can be finished by enforcing the sensitive step (line 12), these two threads are vulnerable (line 13).

5 Implementation

First, a normal user starts crawling the web application sequentially and in-depth but its traffic is saved.

Algorithm 6 Authentication-Bypass-Session-PuzzlingRaceDetection

Input: : CrP as a set of web application log-in critical processes and public-entry point critical processes
At least two valid credentials (CR, CR')
Race windows (RW) of all critical processes in CrP
Output: Detecting if the web application is vulnerable or not

```
1: procedure Authentication-Bypass-Session-PuzzlingRaceDetectionn
2:     Algorithm Authentication-Bypass-Session-PuzzlingRaceDetectionn
3:     Begin
4:      let i=1; // counter for log-in critical processes in CrP
5:      let j=1; // counter for public-entry-point critical processes in CrP
6:      for each log-in critical process in CrP do
7:      let thread1 = log-in critical processi with credential CR;
8:      for each public-entry point critical process in CrP do
9:      let thread2 = public-entry point critical processi with credential CR';
10:     // execute thread1 and thread2 in "race-prone mode" and save results
11:     execute thread1 and thread2 so that WRITE(thread2, CR') in RWthread1;
12:     Refresh current node of thread1
13:      if(current node of thread1 before and after refreshing are not identical)
14:     thread1 and thread2 are vulnerable ;
15:     j++;
16:     done;
17:     i++;
18:     done;
19:     end
20: end procedure
```

According to the user roles, a user can have different levels of access; therefore, depending on the levels of access different user navigation graphs may be obtained. In this research, the normal user has the user access level and navigates the authorized pages. The normal user crawls all different parts of the application that are allowed to navigate. If the user reaches the page without a hyperlink, he/she can return to the homepage of the application and crawl with other hyperlinks.
The user's HTTP traffic is given as input to the BLProM module and the web application business processes are extracted. The business processes identified by BLProM are then examined to select processes with critical race conditions. Business-Layer Session Puzzling Racer uses the MITMProxy server to interact dynamically with the application and receive HTML and JavaScript source files.
Business-Layer Session Puzzling Racer uses the Protractor test framework so that it can load the selected business process in the Google Chrome browser through a proxy and then identify critical processes through the defined conditions. Finally, the results are saved and photographed. In other words, the final page of the process is photographed.

Algorithm 7 Privilege-Escalation-Session-Puzzling-RaceDetection

Input: : BP as a set of business processes of the web application
LGP as a set of web application Log-in processes
At least one valid credentials (CR)
WrP as a set of web application Write processes(that have public entry point or by credential CR)
Privilege-Escalation-Session-Puzzling Race windows (RW) of all log-in processes in LGP
Output: Detecting if the web application is vulnerable or not

```
1: procedure PRIVILEGE-ESCALATION-SESSION-PUZZLINGRACEDETECTION
2:   Algorithm Privilege-Escalation-Session-PuzzlingRaceDetection
3:   Begin
4:    let i=1; //counter for Log-in processes in LGP
5:    let j=1; //counter for write processes in WrP
6:    for each Log-in process in LGP
7:    let thread1 = Log-in processi with credential CR;
8:    for each write process in WrP
9:    let thread2= write processj;
10:   execute thread1 and tread2 so that WRITE(thread2, CR) in RWthread1;
11:   Refresh current node of thread1
12:   if(current node of thread1 before and after refreshing are not identical)
13:   thread1 and thread2 are vulnerable ;
14:   j++;
15:   done;
16:   i++;
17:   done;
18:   end
19: end procedure
```

The identified critical processes are performed twice to assess the vulnerabilities (once in normal mode and in the other time in race-prone mode) and the results are photographed. Photos of each step are compared by using the LooksSame library. If the photos are different, the web application is vulnerable. It should be noted that Business-Layer Session Puzzling Racer does not consider the difference in a pixel (x, y) in both normal and race-prone modes to avoid false positive. Moreover, to determine the sameness of photos in the two modes, some parts of the two photos are displayed in bold.
Business-Layer Session Puzzling Racer can detect the application load completion. The process is executed for each detected business to determine the loading completion of the entire process of pages. Some applications are never loaded because they use a timer and some web application loading is not completed because of the timer. Web applications whose pages have slideshows which are changed automatically every few seconds are a case in point. In such situations, Business-Layer Session Puzzling Racer does not consider timer events that exceed the threshold value when loading the application pages. The assessments showed that the implementation of this policy does not affect the performance of Business-Layer Session Puzzling Racer; however, this policy prevents false positives when photos are compared. Besides, GIF animations are ignored to avoid false positives.

Algorithm 8 Flow-Enforcement-Bypass-Session-PuzzlingRaceDetection

Input: : WrP as a set of write processes of the web application
LGP as a set of web application Log-in processes
SenP as a set of web application sensitive multi-step processes
SimP as a set of web application simple multi-step processes

Output: Detecting if the web application is vulnerable or not

```
1: procedure FLOW-ENFORCEMENT-BYPASS-SESSION-PUZZLINGRACEDETECTION
2:   Algorithm Flow-Enforcement-Bypass-Session-PuzzlingRaceDetection
3:   Begin
4:    let k=1; // counter for simple multi-step processes in SimP
5:    let l=1; // counter for sensitive multi-step processes in SenP
6:    for each simple multi-step process in SimP
7:    let thread1 = simple multi-step processk;
8:    for each sensitive multi-step processes in SenP
9:    let thread2= sensitive multi-step processl;
10:   if(thread1 and thread2 have same length )
11:   execute thread1 except last step;
12:   execute thread2 and pause execution before sensitive step
13:   continue executing thread1 and complete so that WRITE(thread1, flags) in RWthread2 ;
14:   if(thread2 can reach to its final node with skipping sensitive step)
15:   thread1 and thread2 are vulnerable ;
16:   done;
17:   else continue;
18:   j++;
19:   done;
20:   i++;
21:   done;
22:   end
23: end procedure
```

6 Experiment Results and Evaluation

In order to evaluate the proposed method, Business-Layer Session Puzzling Raceris applied to different benchmark applications. The experiments are applied to 6 open-source and well-known applications that may be loaded offline. Table 2 shows the applications used for evaluations; Opencart v3.0.3.3 [46], Ror˙ecommerce [47], Lightning Fast Shop [48], Broadleaf [49], MyBB v1.8.15 [50], Oxid v6.0.2-0 [51] and Puzzlemall.
Our test bed has a web server (test target) and a client (Business-Layer Session Puzzling Racersystem and legal user). Web server and client are loaded on a virtual machine. The web server and the client's profiles are shown in Table 3 . Six applications given in Table 2 are installed on the web server and these web applications are tested in the business layer.

The results of experiments performed on the selected applications are shown in Table 4.

Table 2. Selected web applications for evaluation

Application	Language	Web Deployment in Iran	Web Deployments	GitHub Stars as of 7/12/2020	Lines of Code
Opencart v3.0.3.3 [46]	PHP	1,732	919,041	5,563	136,544
Ror_ecommerce [47]	Ruby (Rails)	NA	NA	1,199	17,224
Lightning Fast Shop [48]	Python	NA	NA	524	25,163
Broadleaf [49]	Java	NA	NA	1,385	163,012
MyBB v1.8.15 [50]	PHP	116	9,520	699	15,689
Oxid v6.0.2-0 [51]	PHP	56	5,899	185	23,126
Puzzlemall	PHP	NA	NA	46	985

Table 3. Test bed profiles

OS: windows 8.1 VMware CPU: 1GHZ VMware RAM: 1G VMware OS: windows 7	Web server (test target)
OS: windows 8.1 VMware CPU: 1GHZ VMware RAM: 1G VMware OS: windows 7	Client (Semantic Web Racer machine)

Table 4 represents the vulnerabilities detected in the selected applications. In Table 4, #attack request shows counter for http requests in the attack scenario for specific vulnerability. #vul (E) shows counter for the vulnerabilities existed in the vulnerable web applications. #vul (D) shows the counter for detected vulnerabilities by Business-Layer Session Puzzling Racer. #TP and #FP shows counter for true positive and false positive of Business-Layer Session Puzzling Racer in detecting vulnerabilities. If any of the selected web application does not have the specific vulnerability, we state "Nothing found" in Table 4. For Client-side Races, we prefer to find them in real web sites because none of the selected web applications were vulnerable to them. As shown in Table 4 the average counter for attack requests where Business-Layer Session Puzzling Racer is applied against the web application is 30.39. It shows that Business-Layer Session Puzzling Racer does not force any considerable traffic to the web application and between 5 existed vulnerabilities in selected web applications and websites, Business-Layer Session Puzzling Racer detects all of them without any false positive.

6.1 Performance evaluations

To evaluate the performance of Business-Layer Session Puzzling Racer, we define the following performance indicators.

- Accuracy: It denotes the precision of Business-Layer Session Puzzling Racer in detecting vulnerabilities.
- Overhead: It is the average CPU usage, memory usage, and bandwidth usage of the Business-Layer Session Puzzling Racer system.
- Effectiveness: It signifies if Business-Layer Session Puzzling Racer is able to detect vulnerabilities in real web applications.

To evaluate the accuracy of Business-Layer Session Puzzling Racer, we used the following metrics: true positive rate (TPR), false positive rate (FPR), false negative rate (FNR), precision, and recall. The TPR, recall, and precision of Business-Layer Session Puzzling Racer are 100%. The FPR and FNR of Business-Layer Session Puzzling Racer are 0%, as they have no false positive or false negative in detecting vulnerabilities. To calculate the overhead of Business-Layer Session Puzzling Racer, we used some metrics such as memory usage, CPU usage, and bandwidth usage of the Business-Layer Session Puzzling Racer system during the execution time for detecting the vulnerabilities of the selected web applications. The average CPU usage, average memory usage, and average bandwidth usage of the Business-Layer Session Puzzling Racer system were 9.60%, 21.99%, and 56.552 KB/s, respectively.

To evaluate the effectiveness of Business-Layer Session Puzzling Racer, we applied Business-Layer Session Puzzling Racer to selected web applications that are widely used in public. As we showed in Table 4 Business-Layer Session Puzzling Racer was able to detect vulnerabilities without any false positive.

6.1.1 Comparison of Business-Layer Session Puzzling Racer and ACIDRain

The main problem of ACIDRain [11] is because it assumes that single-operation transactions cannot create a race condition, it does not recognize all "update race", and "unique-identifier race" items, and most of "limit-based race" items. Another problem with ACIDRain is that after identifying transactions that

Table 4. Detected vulnerabilities in the selected web applications

Vulnerability	Vulnerable Web application	# Attack Requests	# Vul (E)	# Vul (D)	#TP	#FP
Log-in Session Puzzling Race	Puzzlemall	10	1	1	1	0
Authentication-Bypass Session Puzzling Race	Puzzlemall	6	1	1	1	0
User-Impersonation Session Puzzling Race	Puzzlemall	6	1	1	1	0
Priviledge-Escalation Session Puzzling Race	Puzzlemall	6	1	1	1	0
Flow-Enforcement-Bypass Session Puzzling Race	Puzzlemall	11	1	1	1	0

Table 5. Time complexity and overhead complexity of Business-Layer Session Puzzling Racer

Vulnerability	Vulnerable Web application	CPU usage (%)	Memory usage (%)	Bandwidth usage (bps)
Log-in Session Puzzling Race	Puzzlemall	7.21	16.85	16453
Authentication-Bypass Session Puzzling Race	Puzzlemall	7.21	16.85	16453
User-Impersonation Session Puzzling Race	Puzzlemall	7.21	16.85	16453
Privileged-Escalation Session Puzzling Race	Puzzlemall	7.21	16.85	16453
Flow-Enforcement-Bypass Session Puzzling Race	Puzzlemall	9.36	23.56	18798
Average		9.60	21.99	56552

are likely to be abnormal and lead to a race condition, they run a parallel number of transactions to the application to determine whether or not the detected transaction really leads to a race condition. Applying a large number of transactions in parallel to the application increases the likelihood of denial of service attack to the application. The #Nework I/O and #Concurrent Requests criteria were used to indicate the increased likelihood of a denial of service attack.

Figure 13 and Figure 14 compare graphs generated by Business-Layer Session Puzzling Racer and ACIDRain in terms of graph nodes and graph edges, respectively. On average, the generated graph in Business-Layer Session Puzzling Racer has improved about 75.51% in terms of the number of graph nodes and about 73.79% in terms of the number of graph edges compared to ACIDRain. The generated graph in ACIDRain is very large and non-optimal because ACIDRain compares the timestamp of database query and timestamp of API calls. If they are in the same time interval, the desired API calls will generate the database query. In this method, an API can be called multiple times at specific intervals, all of which are displayed independently in the abstract history graph generated by ACIDRain and there is no mechanism to prevent infinite graphs. As a result, the generated graph is not optimal and can be infinitely expanded.

Another problem is that after the identification of abnormal transactions that lead to a race condition, ACIDRain runs a parallel number of transactions to the application to determine if the detected transaction has really led to a race condition. Applying a large number of transactions in parallel to the application increases the likelihood of a denial of service (DoS) attack. Figure 15 and Figure 16 uses the #Nework I/O and #Concurrent Requests criteria to indicate the increased likelihood of denial service attack. Each criterion is explained as follows:

- Network I/O in KB/sec: Network I/O in KB/sec criterion represents the amount of traffic exchanged during the Parallel execution of requests.
- #Concurrent Requests: the number of concurrent requests indicates how many requests are sent in parallel to the web application to make the race condition in the web application.

As shown in Figure 15, the average traffic exchanged during parallel execution of requests in ACIDRain and Business-Layer Session Puzzling Racer is 15038.32 KB/sec and 844.49 KB/sec, respectively. Thus, the Business-Layer Session Puzzling Racer method has improved the exchanged traffic by about 94.38%. The average number of parallel requests sent to an application in ACIDRain and Business-Layer Session Puzzling Racer is 31 and 2, respectively. Thus, the Business-Layer Session Puzzling Racer method has improved the percentage of requests by about 93.52%. Because ACIDRain sends requests when executing requests without considering the time window, it has to send more requests in parallel to create a race condition.

6.2 Comparison of Business-Layer Session Puzzling Racer and ACIDRain

Figure 17 compares the Business-Layer Session Puzzling Racer and the INITRACER in terms of average

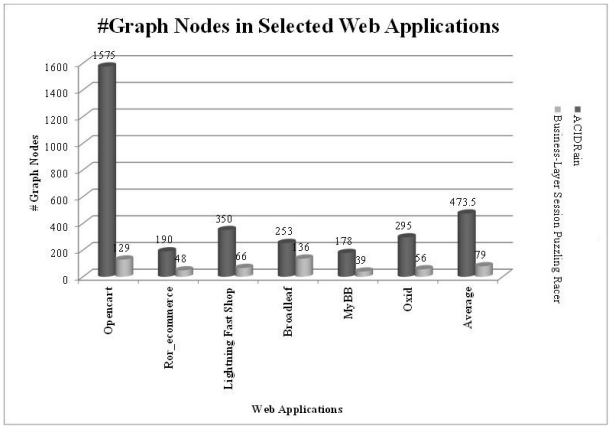

Figure 13. Comparison of average number of graph nodes produced by Business-Layer Session Puzzling Racer and ACIDRain

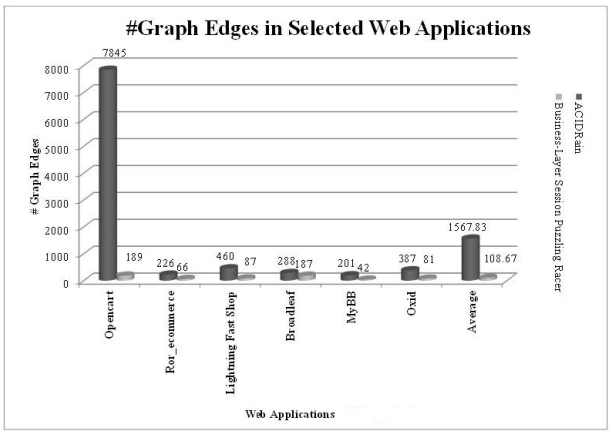

Figure 14. Comparison of average number of graph edges produced by Business-Layer Session Puzzling Racer and ACIDRain

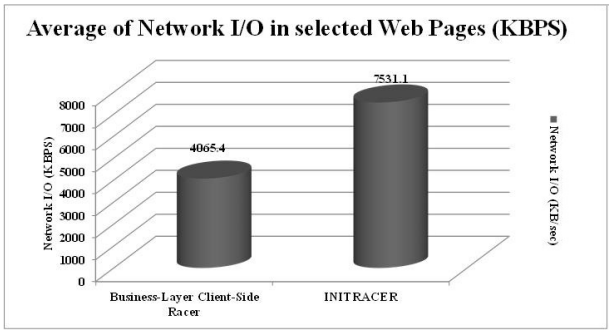

Figure 15. Comparison of average Network I/O in Business-Layer Session Puzzling Racer and ACIDRain

of vulnerability detection time. Business-Layer Session Puzzling Racer goes through three stages; identification of business processes, identification of critical processes, and application of critical processes; to investigate the results. The INITRACER goes three stages; namely, the observation mode, adverse mode, and validation mode. On average, it took 88.81s for the INITRACER to identify the vulnerabilities, while Business-Layer Session Puzzling Racer required, on average, only 23.9s for the same purpose. This is

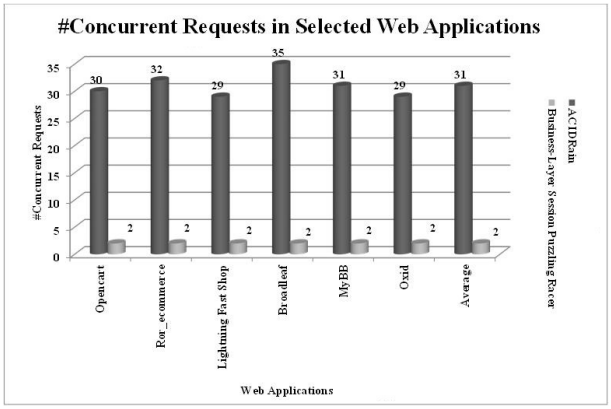

Figure 16. Comparison of average number of parallel requests in Business-Layer Session Puzzling Racer and ACIDRain

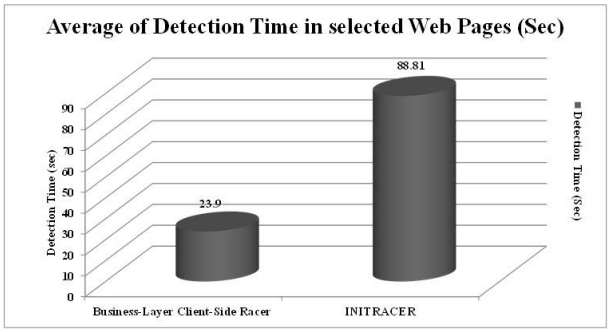

Figure 17. Comparison of average vulnerability detection time in Business-Layer Session Puzzling Racer and INITRACER

rooted in the fact that INITRACER loads the application several times during the identification stage to search for critical sequences, extending the vulnerability identification time and hence lowering its efficiency. In contrast, Business-Layer Session Puzzling Racer receives, as input, the user's HTTP traffic and analyzes it to identify the critical processes. Once the critical process was identified, the application is launched only once in the normal mode and once in the race condition-prone mode.

6.3 Comparison of Business-Layer Session Puzzling Racer and ACIDRain

Figure 18 compares Business-Layer Session Puzzling Racer to AjaxRacer in terms of the time required to identify the vulnerabilities to user-event ajax race condition. The Business-Layer Session Puzzling Racer took an average of 18.93s to identify the vulnerabilities while the AjaxRacer could perform the same in no better than 589.53s on average. This indicates that Business-Layer Session Puzzling Racer could exhibit the same level of accuracy in 96.78% less time. The reason the AjaxRacer detection time is high stems from the fact that for each of the Ajax events in the application, it first generates a separate graph and then evaluates the event for security.

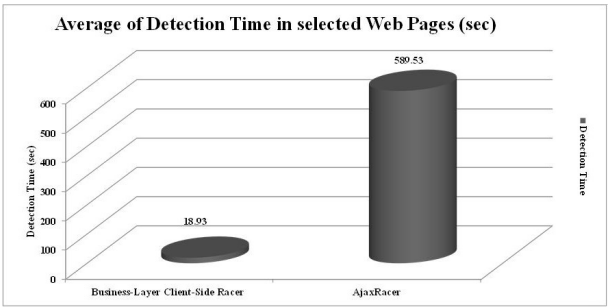

Figure 18. Comparison of average vulnerability detection time in Business-Layer Session Puzzling Racer and AjaxRacer

7 Conclusion

In this study, various session puzzling race conditions are studied and types of session puzzling race conditions existing in the applications are defined. Moreover, the black-box method (Business-Layer Session Puzzling Racer) is proposed for the dynamic application security testing of the web application in the business layer. The proposed approach detects business-layer vulnerabilities of the web application against various session puzzling race conditions. Furthermore, Business-Layer Session Puzzling Racer does not result in DoS attack on the web application and improves the vulnerability detection traffic by about 94.38%. In future works, we intend to extend the proposed approach to identifying server-side and client-side race conditions with a view to detecting various types of race conditions.

References

[1] Mitra Alidoosti, Alireza Nowroozi, "BLProM: Business-layer Process Miner of the web application", International Conference on Information Security and Cryptology, 2018.

[2] M. Alidoosti, A. Nowroozi, A. Nickabadi, "BLProM: A Black-Box Approach for Detecting Business-Layer Pro cesses in the Web Applications", Journal of Computing and Security, vol.6, no.2, pp.65-80, july 2019.

[3] M. Alidoosti, A. Nowroozi, and A. Nickabadi, "Evaluating the Web-Application Resiliency to Business-Layer DoS Attacks," ETRI Journal , vol.42, no.3, 2019. doi:10.4218/etrij.2019-0164.

[4] E. Brattli Sørensen, Jingyue Li, "A Literatu re Review and Practitioner Survey on Using Vulnerability Detection Tools to Defend Against Access Control Vulnerabilities," Technical Report, Department of Computer Science, Norwegian University of Science and Technology, December 2019

[5] S. Zeller, N. Khakpour, D. Weyns, D. Deogun, "Self-Protection Against Business Logic Vulnerabilities," IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Seoul, South Korea, May 2020.

[6] F. Nabi, J. Yong, and X. Tao, "A Novel Approach for Component based Application Logic Event Attack Modeling," International Journal of Network Security, vol.22, no.3, pp.437-443, May 2020.

[7] Y. Chen, L. Xing, Y. Qin, X.Liao, X. Feng Wang, K. Chen, W. Zou, "Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis," In28th USENIX Security Symposium Security, pp. 747-764, 2019.

[8] M. Ghorbanzadeh, HR. Shahriari, "Detecting application logic vulnerabilities via finding incompatibility between application design and implementation," IET Software, vol. 14, no. 4, pp. 377-88, Mar 2020.

[9] H. Homaei, HR,. Shahriari, "Seven years of software vulnerabilities: The ebb and flow," IEEE Security & Privacy, vol. 15, no. 1, pp. 58-65, Feb 2017.

[10] H. Homaei, HR,. Shahriari, "Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour." Information and Software Technology, vol. 1, no. 107, pp. 112-24, Mar 2019.

[11] M. Monshizadeh, P. Naldurg, VN. Venkatakrishnan, "Vulnerabilities for web applications using logic patcher," In Sixth ACM Conference on Data and Application Security and Privacy, pp. 73-84, Mar 2016.

[12] Deepa G, Thilagam PS, Praseed A, Pais AR, "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications. Journal of Network and Computer Applications," Vol.109, no.1, pp. 89-109, May 2018

[13] RJ. Emous, "Towards systematic black-box testing for exploitable race conditions in web apps", Master's thesis, University of Twente.

[14] R. Paleari, D. Marrone, D. Bruschi, M. Monga. "On race vulnerabilities in web applications". In-International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment , Springer, Berlin, Heidelberg. pp. 126-142, July 10 2008.

[15] W.G. Halfond , J. Viegas , A. Orso,: "A Classication of SQL-Injection Attacks and Countermeasure"s. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA ,March 2006.

[16] CERT: Advisory CA-2000-02: "Malicious

Table 6. Comparing related works

Research Article	Analysis type	Analysis approach	Type of vulnerability	Type of approach	Weakness
INITRACER [34]	Dynamic	Three phase detection	Only 3 race vulnerability	automatic	Only Ajax races during web page initialization
CompuRacer [13]	Dynamic	BY sending parallel request	Has checklist of vulnerabilities	Systematic not automatic	DoS attack and make web application unavailable
AJAXRacer [33]	Dynamic	By triggering special Ajax events in two modes and comparing them	Only 3 category of races	automatic	Only Ajax races after web page initialization
WebRacer [28]	Dynamic	By detecting happens-before relations	Only some of Ajax races	automatic	Only formally define happens-before relations
ARROW [35]	Static	By detecting happens-before relations and Def-use relations	Only some of Ajax races	automatic	It has false positive
RClassify [36]	Static	Recognizing harmful races from receiving alerts	Only some of Ajax races	automatic	It only classifies harmful and harmless races
Paleari *et al.* [14]	Dynamic	By monitoring interaction between database and web application	Only data base races	automatic	It needs to define proper input and interleaving
EventRaceCommander [37]	Static	Repairing Ajax Races by postponing user/system events	Only Ajax races	automatic	It has false positive
Mutlu *et al.* [32]	Dynamic	Analyzing dataflow of sensitive variable in different scripts	Only one Ajax race	automatic	Ignore harmful races in web application initialization
EventRacer [39]	Dynamic	Analyzing happens-before relations and defining vector clock for events	Only four Ajax race	automatic	Reports over-whelming counter for harmless races

Table 7. Comparing previous studies in terms of detecting vulnerabilities

	Log-inSessionPuzzlingRace	Authentication-BypassSessionPuzzlingRace	User-ImpersonationSessionPuzzlingRace	Privilege-EscalationSessionPuzzlingRace	Flow-Enforcement-BypassSessionPuzzlingRace
INITRacer [34]	×	×	×	×	×
CompuRacer [13]	√	×	×	×	×
AJAXRacer [33]	×	×	×	×	×
WebRacer [28]	×	×	×	×	×
ARROW [35]	×	×	×	×	×
RClassify [36]	×	×	×	×	×
Paleari *et al.* [14]	×	×	×	×	×
EventRaceCommander	×	×	×	×	×
Mutlu *et al.* [37]	×	×	×	×	×
EventRacer [32]	×	×	×	×	×
DetLogic [41]	×	√	√	×	×
Business-LayerSessionPuzzlingRacer	√	√	√	√	√

HTML Tags Embedded in Client Web Requests" ,2002.

[17] Netzer RH., Miller BP., "What are race conditions?: Some issues and formalizations", Programming Languages and Systems, vol.1, no.1, pp.74-88, 1992.

[18] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multihreaded C++ programs", Proceedings of the Symposium on Principles and Practice of Parallel Programming, June 11-13, 2003, San Diego, Canada.

[19] D. Dean and A. J. Hu, "Fixing races for fun and profit: how to use access(2)", USENIX Security Symposium, vol.2, no.14, 2004.

[20] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses", USENIX Security Symposium, vol.17, 2003.

[21] M. Bishop and M. Dilger, "Checking for Race Conditions in File AccessesComputing Systems", vol. 9, no. 2, pp. 131-152, 1996.

[22] PA. Emrath, S. Ghosh, DA. Padua, "Detecting nondeterminacy in parallel programs". IEEE Software, vol.9 no.1, pp:69-77, Jan 1992.

[23] C. Flanagan, SN. Freund, "Detecting race conditions in large programs". InProceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 90-96, Jun 2001.

[24] SV.Adve, MD. Hill, BP. Miller, RH. Netzer, "Detecting data races on weak memory systems". ACM SIGARCH Computer Architecture News.

Vol. 19, no.3 , pp. 234-243, May 1991.

[25] Shin Hong, Yongbae Park, and Moonzoo Kim. "Detecting Concurrency Errors in Client-Side JavaScript Web Applications".In Proc. 7th IEEE International Conference on Software Testing, Veriication and Validation, 2014.

[26] James Ide, Rastislav Bodik, and Doug Kimelman." Concurrency Concerns in Rich Internet Applications. In Proc. Workshop on Exploiting Concurrency Eiciently and Correctly, 2009.

[27] CS. Jensen, A. Mùller, V. Raychev, D. Dimitrov, and M.T. Vechev. 2015. "Stateless Model Checking of Event-Driven Applications". In Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Pro-gramming, Systems,. Languages, and Applications , 2015

[28] B. Petrov,M. Vechev, M. Sridharan, J. Dolby. "Race detection for web applications". InACM SIGPLAN Notices , Vol. 47, No. 6, pp. 251-262, June 2012.

[29] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster. "ARROW: Automated Repair of Races on Client-Side Web Pages". In Proc. 25th International Symposium on Software Testing and Analysis , 2016.

[30] Y. Zheng, T. Bao, and X. Zhang, "Statically Locating Web Application Bugs Caused by Asynchronous Calls". In Proc. 20th International Conference on World Wide Web, 2011.

[31] V. Raychev, M. Vechev, M. Sridharan. "Effective race detection for event-driven programs". InACM SIGPLAN Notices ,Vol. 48, No. 10, pp. 151-166,. October 2013

[32] E. Mutlu, S. Tasiran, B. Livshits. "Detecting JavaScript races that matter". InProceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 381-392, August 2015.

[33] CQ . Adamsen, A. Møller A, F. Tip. "Practical AJAX race detection for JavaScript web applications". InProceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018.

[34] CQ . Adamsen, A. Møller A, F. Tip. "Practical initialization race detection for JavaScript web applications". Proceedings of the ACM on Programming Languages. Vol. 1, issue. OOPSLA, October 2017.

[35] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster. "ARROW: Automated Repair of Races on Client-Side Web Pages". In Proc. 25th International Symposium on Software Testing and Analysis , 2016.

[36] L. Zhang, C. Wang. "RClassify: classifying race conditions in web applications via deterministic replay". InProceedings of the 39th International Conference on Software Engineering , pp. 278-288, 2017.

[37] CQ. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, K. Sen. "Repairing event race errors by controlling nondeterminism". In2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) ,pp. 289-299, May 2017.

[38] S. Chen." Session Puzzling and Session Race Con-ditions", 2011. [Online]. Available: http://sectooladdict.blogspot.com/2011/09/session-puzzling-and-session-race.html.

[39] R. Veselin, M. T. Vechev, and Manu Sridharan. " Efective Race Detection for Event-Driven Programs". InProc. 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, 2013.

[40] S. Ramachandran. "Web metrics: Size and counter for resources". https://developers.google.com/speed/articles/web-metrics. Last updated: 26 May 2010.

[41] Deepa G, Thilagam PS, Praseed A, Pais AR. DetLogic: A black-box approach for detecting logic vulnerabilities in web applications. Journal of Network and Computer Applications. Vol.109, no.1, pp. 89-109, May 2018.

[42] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb, "Security Analysis and Enhancements of Computer Operating Systems," National Bureau of standards Washington, D.C., Technical report, 1976.

[43] M. Jadon. (2018) Race Condition Bug In Web App: A Use Case. [Online]. Available: https://medium.com/@ciph3r7r0ll/race-condition-bug-in-web-app-a-use-case-21fd4df71f0e.

[44] W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, no. 3, pp. 208–215, 1976.

[45] Hallvord Reiar Michaelsen Steen. 2009. Websites playing timing roulette. https://hallvors.wordpress.com/2009/ 03/07/websites-playing-timing-roulette/. (2009).

[46] OpenCart. https://www.opencart.com/.

[47] ror˙ecommerce. https://github.com/drhenner/ror_ecommerce.

[48] Shop, Lightning Fast. https://github.com/diefenbach/django-lfs.

[49] Commerce, Broadleaf. https://github.com/BroadleafCommerce/BroadleafCommerce.

[50] MYBB. https://mybb.com/.

[51] oxid. www.oxid-esales.com.

Mitra Alidoosti received the B.S. and M.S. degrees in computer engineering from the Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran, in 2009 and 2012, respectively. Currently, she is working toward the Ph.D. degree in computer engineering at Malek-e-Ashtar University of Technology, Tehran, Iran. Her research interests are computer network security, VoIP and SIP security, and web-application security.

Alireza Nowroozi is a freelance consultant who advises government and private-sector-related industries on information technology. He has four-year experience as an academic staff and an IT post-doctoral position with Sharif University of Technology, Tehran, Iran. He is a specialist in artificial intelligence, cognitive science, software engineering, and IT security, and he is a co-founder of four IT startups.

Ahmad Nickabadi received the B.S. degree in computer engineering in 2004, and the M.S. and Ph.D. degrees in artificial intelligence in 2006 and 2011, respectively, from Amirkabir University of Technology, Tehran, Iran. He is currently an assistant professor with the Department of Computer Engineering, Amirkabir University of Technology. His research interests include statistical machine learning and soft computing.